Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

# TYPES FOR LAZY LOGIC PROGRAMS: COMPUTATION AND INFINITE OBJECTS

S.-T. Huang
D. S. Parker

June 1992
CSD-920029

**Types For Lazy Logic Programs : Computation and Infinite Objects** [1]

Shen-Tzay Huang and D. Stott Parker

sthuang@cs.ucla.edu stott@cs.ucla.edu

Computer Science Department

University of California

Los Angeles, CA 90024-1596 USA

**Abstract**

A framework for types of logic programs, in particular, those embodying lazy computation and infinite objects, is proposed. Examples are given to illustrate the need. Starting with two methodological views, **Types As Sets** (subsets of the Herbrand Universe) and **Types for The Prolog Computational Model**, we give a simple monomorphic type structure and extend it with principles of polymorphism and homogeneity. The SVP model of Parker, Simon and Valduriez is given a detailed type analysis as an application of this framework. Common logic programs serve as other instances of application. Directions for enriching the type structure (Herbrand Type Universe) are discussed.

[1] This research is supported by NSF grant IRI-8917907

# Contents

1 **Introduction** 4

2 **Types in Logic Programs** 4

3 **The Herbrand Value Universe and Herbrand Type Universe** 6

3.1 Type Structure . . . . . . . . . . 6

3.2 Principle of Polymorphism and Homogeneity . . . . . . . . . . 8

4 **Applications** 10

4.1 Type Specification of SVP Transducer Scheme . . . . . . . . . . 10

4.1.1 The SVP Model in The Herbrand Type Framework . . . . . . . . . . 10

4.1.2 Types of SVP Transducers . . . . . . . . . . 12

4.1.3 Summary . . . . . . . . . . 13

4.2 Another Application . . . . . . . . . . 14

5 **Conclusion and Further Research** 14

# 1 Introduction

As a doctrine of first order logic, expressions are naturally classified into three categories, namely, terms, predicates and sentences. By this, the objects of discourse, namely, (logical) terms, are simply treated as symbolic objects. They bear no meaning, except denoting themselves, a string from a given alphabet. The question of classifying or structuring the set of all terms does not carry much significance theoretically, since it amounts to straightforward restriction in the category of well-formed formulas, including predicates and sentences. However, in practice, as notations do carry pragmatic significance, we do want to impose structure, particularly that under the umbrella of types, directly on the terms. In addition, types help not only to correct reasoning, say, Russell's paradox, but in practice, serve as an indispensable part of program specification and correct implementation in programming. It is important to have a type scheme for any logic-based system, in particular, the Logic Programming paradigm [SS 87].

Many approaches have been proposed to deal with the issues of types in Logic Programming, [YS 91, XW 88, GM 84, Mi 84]. In this article, we focus on the issue of types for lazy computation and uninstantiated logical variables, which we think has not yet been satisfactorily resolved. For the study, we take two methodological stands, **Types As Sets** and **Types for The Prolog Computational Model**, due to the framework and applications we have in mind. These issues have direct implications for stream processing in Logic Programming.

This article is organized in five sections. Section 2 gives examples of our interests, and argues for the need of a type scheme different from those commonly borrowed from functional programming. In section 3, by respecting the dynamic nature of computation and adapting existing theoretical framework on semantics of Logic Programs, we propose a type scheme built on a **Herbrand Value Universe**, similar in nature to the Herbrand Universe. We also discuss extensions with polymorphism and homogeneity to the simple monomorphic type structure, the **Herbrand Type Universe**, an instance of the type scheme. In section 4, we apply the framework to a type analysis of a particular intended domain, the SVP model [SVP 92]. The analysis also gives concrete illustration of the ideas and technique behind the general framework. We also apply it to the problems raised in section 1. In the final section, we give conclusions of the study and directions for further research.

# 2 Types in Logic Programs

As its underlying foundation is untyped, Logic programming suffers an obvious inherent disvantage of being prone to type errors. As a frequently offered example :

**Example. 1**

$$
\begin{array}{lll}
append([], L, L) & :- & true. \\
append([X|Xs], Ys, [X|XsYs]) & :- & append(Xs, Ys, XsYs).
\end{array}
$$

The case in point is when the goal, $?-append([], a, X)$, is invoked, a non-list binding of X is returned successfully, which conflicts with the intension of this program. In general, such cases of unintended binding of incorrect type will occur as long as there are two or more kinds, or types in formal terms, of atomic constants. Unfortunately, there is no way to prevent this from happening, as Logic programs are inherently untyped, with computations on a nominal canonical first order term universe, the **Herbrand universe**, where all values are of the single type **Terms**.

As another example, the logical variable X of the first clause below, could be instantiated with any term denoting a non-natural number, and yet still attain successful execution with a correct answer substitution that respects the formal semantics of the program.

**Example. 2**

$$\begin{array}{lll} plus(0, X, X) & :- & true. \\ plus(succ(X), Y, succ(Z)) & :- & plus(X, Y, Z). \end{array}$$

To avoid these unwanted pitfalls, conventional type methods for logic programs usually introduce declarations of type scheme or type predicates to enforce type consistency, for instance, $list(L)$ and $nat(X)$ for the two examples, respectively.

Plausible as these methods are, we don't think they give a satisfactory solution, in particular for Logic Programs per se, for the following reasons :

- While these type predicates are primarily introduced to enforce a static type discipline, they are added to the body of original programs. So, in effect, run-time checking is still performed. Although they do enforce disciplined programming, these type predicates add more computational cost, as they themselves are described as logic programs, indistinguishable from other clauses to the Prolog engine. This drawback is due to the direct adoption of type methods used in common procedural or functional programming languages. While it appears as a necessary evil due to the first-order nature of logic programs, the price is high enough to the lose distinctive features of the Logic Programming paradigm. We shall elaborate on this point later.
- Even in the above two examples, as commonly typed by $list/1$ and $nat/1$ respectively, there is a difference from functional implementations in LISP. For example, $nat/1$, or the type of natural numbers, is homogeneous and monomorphic globally. That is, we would expect any value and its subcomponents thus typed would be the same everywhere. On the other hand, this is not quite the case with $list/1$. In LISP, we are allowed to have $cons(a, X)$, where $X$ is an arbitrary atom other than nil, []. Of course, we can have this in Prolog, but this is not captured in the common type predicate-based methods and it could be arguable if this is still a "list". Furthermore, [] is actually a constant of polymorphic type $list(T)$, while the atom, 0, is of monomorphic type $nat$. This particularly deviates from the essence of nominal term models where each ground term denotes a value (itself). The ground term [] is now treated as an infinite scheme of ground terms, for example, the integer list [], rational list [], character list [], etc.
- In addition to these problems with polymorphic ground terms, lists also demonstrate the difficulties when logical variables are used, especially, when uninstantiated variables corresponding to laziness in computation or placeholders are used. There doesn't appear to exist any reasonable way to assign a type to an uninstantiated logical variable, based on commonly-used type systems, without introducing a great deal of complication or losing the intuition provided by logical variables per se.
- In actual programming, some instances of values and operators are indeed less sensitive to type constraints, such as sets of entities (of arbitrary types), operators like dequeue() and enqueue() of entities (of arbitrary types). While it is arguable whether powerful type schemes, e.g. the second order $\lambda$-calculus [Mi 90], can be employed to cope with type constraints, they pose no problem for the Logic Programming paradigm. Furthermore, it doesn't appear that a powerful type scheme, such as the second order $\lambda$-calculus and its variants, is needed for logic programs implementing real applications. For instance, function types are not (commonly) used in Logic programs, as functions are modeled as predicates.

In the above discussion, we emphasize the use of logical variables, allowance of lazy computation and a certain degree of heterogeneously typed or untyped terms as distinct features of logic programs in comparison with conventional procedural and functional paradigms. These are also features we want to maintain, after types are introduced. And applications requiring such features are not uncommon, for example, difference lists, infinite objects, stream processing [SVP 92], where logical variables are incrementally but never fully instantiated.

As the aim is set, we find there are two issues anchored on the notion of Logic Programming as a paradigm with Prolog as a key instance.

As often condensed in the slogan **Algorithm = Logic + Control**, declarativeness is emphasized as a key advantage over proceduralness. And Logic Programming bears both declarative and procedural readings in a modular style. As operational semantics, model theoretic semantics and denotational semantics can be identified for logic programs, the procedural and declarative distinction no longer exists, at least theoretically. However, SLD resolution is used as a computational counterpart for declarative logic expressions, and many extralogical mechanisms are embodied in Prolog. We have to ask a question again, in the context of types, as follows :

**Is the notion of type intended for 'Prolog as a (logic) programming language'**

**or for 'Prolog as a computational model' ?**

The issue of course resembles **Logic vs. Control**, or **Declarative vs. Procedural**. The former seems to make the notion of types static, while the latter dynamic. Logical variables and lazy evaluation are more dynamic in essence and in practice; and untyped terms are both static and dynamic in the context of Herbrand models and related identification theorems about semantics. Without detailed elaboration, simply as a commitment, we choose to deal with types in the context of **'Prolog as a computational model'**, in brief, **Types for The Prolog Computational Model**. This implies we are interested in how types interact and evolve from a procedural perspective, in short, the dynamic aspects of objects in computations. Doubtless, this will affect our methods for inquiry. So far, we would only justify this commitment by the results and applications we are pursuing, but we do believe this is a direction worth pursuing within the Logic Programming community.

# 3 The Herbrand Value Universe and Herbrand Type Universe

We assume some familiarity with notions and terminologies of logic programming [SS 87] and its foundations [Ll 87]. As for the general question about **What Is A Type**, we take **A Type Is A Set**, rather than others like **Type as Formula, Type as Propositions**, [Ho 80, FLO 83, CW 85, Mi 90]. This is due to our applications where no function space is required and consequently there are no concerns of self-reference and impredictability, on the one hand, and has the foundational advantage that a canonical semantic model of the Herbrand Universe is given as a common domain to build type schemes, on the other. In particular, the subsets of the Herbrand Universe can serve as a meaning of **Type**, and thus **A Type Is A Set**. Also, we don't have to worry about whether the set of types is a type, namely, $\mathbf{TYPE} \overset{?}{\in} \mathbf{TYPE}$. It is obvious that the set of all types need not be a type in our framework. This approach is in line with the ideas of general models for higher order logic $\mathcal{L}_\omega$, by Hintikka as explained in the survey [BD 83]. In that framework, a type structure is imposed over a first order structure, by interpreting higher order terms and variables as higher order functions over the first order structure. Thought similar in terms of idea, our structure, Herbrand Type Universe over Herbrand Value Universe, is much simpler, since function space is not of concern here.

## 3.1 Type Structure

**Definition. 1 (Herbrand Value Universe)** *The Herbrand Value Universe, $H_{Value}$, is defined as the set of value terms inductively constructed from :*

- *any 0-ary constant symbol is a value term.*

- *if $f$ is an n-ary value constructor and $t_1, \cdots, t_n$ are value terms, then $f(t_1, \cdots, t_n)$ is a value term.*

**Remark 1.** Essentially, the Herbrand Value Universe is the usual Herbrand Universe constructed from functors, so-called term constructors. The difference is the restriction to a special subset of functors, called value constructors, interpreted as data constructors. This is only a slight deviation from term models of Logic Programs, where each functor is treated as a value constructor. Here, general functors, other than value constructors, are uninterpreted, sharing the same status as Skolem constants. Our approach would require a restoration of interpretation of these functors as denoting functions on the Herbrand Value Universe. Interpretations, both of value constructors and functors, are of course application-dependent. On the other hand, we can think of these functors as interpreted as functional predicates, having a type distinct from terms. Our method is to downgrade them back into terms, specifically into value terms.

**Definition. 2 (Herbrand Type Universe)** *The Herbrand Type Universe, $H_{type}$, is defined as the union of, the set of type terms inductively constructed from :*

- *the symbol* **atom** *is a type term.*
- *if $f$ is an n-ary type constructor and $t_1, \cdots, t_n$ are type terms, then $f(t_1, \cdots, t_n)$ is also a type term,*

*and, the singleton set* {**Utype**}, *where* **Utype** *is the type of all Herbrand Value Universe.*

**Remark 2.** Intuitively, we use a two-level approach to impose a type structure over values, by classifying the common Herbrand Universe into subsets. The type **atom** contains all primitive constants, and usually this is referred to as a basic type. There actually could be many basic types. The **Utype** embodies all values, with denotation the whole Herbrand Value Universe. One of the import of **Utype** is to wrap up terms which is correct with respect to Prolog model-theoretic semantic yet eluded from proper type assignment.

**Remark.3.** While we leave open non-value-constructing functors as interpreted domain-specific functions defined over value terms, we don't have function abstraction in our framework, in general. So overall, so far, ours is a fragment of the simple type theory [Ch 40]. However, as we shall show, this simple basis can be enriched with other concerns, such as polymorphism, homogeneity, type unification, etc, to serve our goal of modeling uninstantiated logical variables and lazy evaluation. To be precise, uninstantiated logical variables are modeled as being of type **Utype**, i.e., as being unrestricted with respect to the Herbrand Value Universe. Lazy computation is captured in incremental type and value instantiation of logical variables, corresponding to the typed place holders and value assignment of conventional variables.

Note there is a correspondence between type structure and value structure. Precisely, we have the following formal results.

**Proposition. 1** *There is a homomorphic mapping from the Herbrand Value Universe to the Herbrand Type Universe, defined as :*

$$\mathcal{H} \quad : \quad H_{Value} \mapsto H_{Type}$$

$$\begin{array}{lcll} \mathcal{H}(a) & = & \textbf{atom} & \textit{if } a \in \textbf{atom} \\ \mathcal{H}(d_n(X_1, \cdots, X_n)) & = & d_n\text{-}type(\mathcal{H}(X_1), \cdots, \mathcal{H}(X_n)) & d_n \textit{ is n-ary and} \\ & & & type\text{-}constrain(d_n, \mathcal{H}(X_1), \cdots, \mathcal{H}(X_n)) \\ \mathcal{H}(d_n(X_1, \cdots, X_n)) & = & \textbf{Utype} & \textit{otherwise} \end{array}$$

The $type - constrain()$ represents further constraints among constituent types. In its simplest case, the constraint is just a match with the (type) signature of functor $d_n$. In most of our intended applications,

the constraint would be $\mathcal{H}(X_1) = \cdots = \mathcal{H}(X_n)$ encoding the homogeneity of the constituents, common to aggregation object. Such application specific constraints would simplify the formulation of above-posed mapping. For instance, the list concatenation operator, ++, has a constraint over its two constituent operands to be lists with same elementary type, $T$ say. So the value term $(+ +_2 (L_1, L_2))$ would be type-assigned with $list(\mathcal{H}(L_1)) = list(T)$, with type constraint $\mathcal{H}(L_1) = \mathcal{H}(L_2)$, and **Utype** otherwise. Other kinds of constraints and formulations, such as $list(\mathcal{H}(L_1) \cap \mathcal{H}(L_2))$ if exists w.r.t. underlying type structure, further points to interesting aspects enabled in the extension of the Herbrand Type Universe. In general we call the mapping $\mathcal{H}$ a value/type assignment.

**Proposition. 2** *All type terms in the Herbrand Type Universe are ground, representing monomorphic types, and their denotations are mutually exclusive subsets in the Herbrand Value Universe. Furthermore, the union of denotations of non-***Utype** *types is a proper subset of Herbrand Value Universe. In short, every type is a subtype of* **Utype**.

In summary, we have a basic type structure on the Herbrand Type Universe as depicted in the figure :

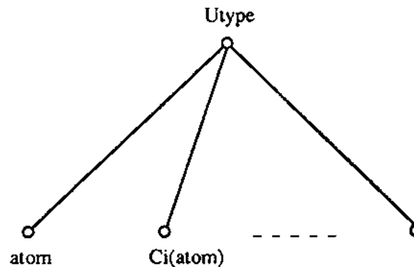


Figure 1: Structure of The Herbrand Type Universe

(In fig. 1, Ci() stands for a type constructor.)

As easily seen, this type structure is in fact a complete join semilattice with ordering as set inclusion, though not richly structured, say, in comparison with the full type structure for second order $\lambda$-calculus. This granted, and given familiarity with Prolog and its foundations, in the discussion in later sections, we will simply assume trivial lattice operators and properties and take a degree of freedom with related terminology (for example, ordering, notion of subsumption, and greatest lower bound as unification).

## 3.2 Principle of Polymorphism and Homogeneity

So far, we have imposed a basic type structure over objects of computation in the Logic Programming paradigm. Yet, it is only monomorphic and only for ground terms, corresponding to a fragment of simple type theory [Ch 40] for functional programming. In the following, we will introduce polymorphism which takes logical variables into consideration.

Basically, polymorphism refers to the case where a single operator can be applied to multiple types of objects. Moreover, polymorphism could refer to objects which can belong to multiple value domains. Whereas there is no need of such a distinction in functional programming, due to the inclusion of higher order functions as objects, having nominal terms as values in logic programming does raise such minor distinctions as those shown by the term **nil**, []. Furthermore, as a consequence of **Prolog Computational**

**Model**, type-unrelated or type-incorrect terms can be computed dynamically even in a typed context. **And** with the position that **A Type Is A Set** of terms, in addition set-theoretic level of being typed, we need a constraint of homogeneity, capturing the ideas of uniformity of constituents at the elementary (value) level, over these sets, to enforce a structure on the set denoted by a type. Later examples will clarify this point make the notion of polymorphism and homogeneity concrete in terms of the type system we have proposed so far.

In general, polymorphism of transducers and mappings is surely a desirable feature, for computational models like that in [SVP 92]. At the same time we want the notion of homogeneity for values and mappings. Intuitively this would prevent us from having either complicated mapping or synchronizing some type information in the specification or actual codes and computations, but homogeneity is a natural and assumption for most practical logic programs, especially those bearing a sense of aggregation in their underlying domain and major operations, such as lists.

The introduction of polymorphism naturally brings with it the notion of quantified type variables, and thereby, quantified type expressions. In other words, polymorphism can be easily modeled with (meta-) logical/type variables ranging over all possible elements of the Herbrand Type Universe. As an obvious consequence, the notion of ordering of polymorphism, issues of subsumption and instantiation can be modeled as unification on the Herbrand Type Universe. But the issue of quantification is tricky, in particular regarding its scope, which potentially leads to a spectrum of notions of polymorphism. In our framework, we would characterize our notion of polymorphic types in this type system with prenex normal form type expressions whose type variables are all universally quantified. By this, we disregard the issue of whether existential variables are allowed, which is considered important for the account of abstract data types. For theoretical concerns about such design choices, see related discussions in [CW 85, Mi 90]. Not to exclude other alternatives, we state our choice as a **Principle of Polymorphism and Homogeneity**. We clarify the ideas and considerations by some examples.

The polymorphic type, $\forall T.List(T)$ is homogeneous as it instantiates all elements in a list with same type. On the other hand, $List(\forall T)$ could instantiate each element with different types, and then give us a heterogeneous, if not chaotically typed, list. Both instances have their applications, for example, the former is commonly used in most list processing, and the latter can be used to model **sets or bags** of objects. As another example, the type prescription of list concatenation, *append()* in our first example, usually is typed as $\forall T.List(T) \times List(T) \rightarrow List(T)$, though $\forall T_0.List(T_0) \times \forall T_1.List(T_1) \rightarrow \forall T_2.List(T_2)$ would do too.

Sorting is another example frequently referred to for illustrating the idea of polymorphism in homogeneous mode. In general, a sorting algorithm will work for any type of objects. Usually, values from a common monomorphic set are stored and referenced indirectly by pointers, and comparisons between these values are employed to decide the order of values and to rearrange their pointers accordingly. Here the ordering is homogeneous, and is the only necessary information, shared between values. The essence is that all objects to be sorted are related, in the sense that they are not only elements of a common set, but also are homogeneously related by a partial ordering.

On the other hand, there are computations which are really fully heterogeneous; that is, they are not dependent on the common structure or value domain of the processed objects. For instance, cardinality or counting processes care only about the number of occurrences, whichever domains they are from. In the most general case, they can be specified as finite-set$(\forall T) \mapsto \mathcal{N}$, where the type variable quantifier is not in prenex normal form.

Though these examples of heterogeneity can be homogenized through an implicit type coercion process, such an interpretation would imply extra program code and type notation for **'Prolog as a (logic) programming language'**, and this does not respect the dynamic process of **The Prolog Computational Model**.

As a remark, we notice that the notions of polymorphism and homogeneity are closely related, though

not identical in the context of logic programming, and both depend on the data representation and the operations to be applied. As a result, any scheme of type assignment should take such pragmatic aspects into account. Lazy computations and infinite objects require such considerations.

In summary, we take the view that polymorphism corresponds to the introduction of type variables in the type specification, and aspects of homogeneity enforce the range of possible types semantically and universally quantified prenex normal form for type expressions syntactically. Furthermore, the underlying type universe is the Herbrand Type Universe with **Utype** capturing heterogeneity. The result is also a predicative type system, in addition to being complete join semilattice depicted in fig. 1, like Martin-Löf's type system [Ma 73], excluding the type of all types from its type structure.

# 4 Applications

In this section, we apply the type framework to a stream processing model, **SVP** [SVP 92], in particular, type analysis of the general scheme of SVP-transducers. This application covers most of our intent for developing the type scheme for lazy computation and infinite objects in logic programming. Additional concerns in this application would be formalized and added as type constraints. We detail an example to further illustrate the ideas and notions in our type framework discussed above. A second application deals with the type assignment of our first example, the logic program *append*/3.

The type inference scheme we use is much simpler than the implicit typing in section 4.6.2 of [Mi 90]. This is because we don't need rules of abstraction and application, since there is no use of function spaces in this application yet. In accordance with the Herbrand type framework, the principle of polymorphism and homogeneity is assumed. So every type variable is universally quantified in prenex normal form, and ranges over the Herbrand Type Universe generated from basic types and type constructors. Furthermore, any solution type assignment is based on unification of type terms positionally, postulated as type equations. As a result, uninstantiated type variables can be assigned to any term, and type constraints can be solved as incremental instantiation of type variables over the monomorphic complete join semi-lattice Herbrand Type Universe.

## 4.1 Type Specification of SVP Transducer Scheme

### 4.1.1 The SVP Model in The Herbrand Type Framework

SVP is a data model, capturing both set and stream data, and is designed to model parallelism in bulk processing. The principle data construction is *collection*, of which the set and stream are special cases. It also allows parallelism of many database queries to be captured in the format of divide-and-conquer mappings, when specified using collections.

The collection data model and associated general scheme for SVP-transducers fit very well in our scheme. And we also find the type analysis helps not only validate claims but also refine specifications.

In SVP, the corresponding Herbrand Value Universe is the set of SVP values, constructed from basic values and value constructors, *tuples()* and *collection()*. Formally, SVP values are recursively defined as :

- Any atom is SVP value. We call these as SVP basic values.
- Any finite tuple $(v_1, \cdots, v_n)$ of SVP values $v_1, \cdots, v_n$ is a SVP value.
- Any collection is a SVP value.

The value of *collection* is defined as :

- $<>$ is the empty collection.
- $< v >$ is a unit collection if $v$ is a SVP value.
- $S_1 \diamond S_2$ is a collection if $S_1$ and $S_2$ are nonempty collections.

Note, in effect, the Herbrand Value Universe contains some non-SVP values according to the definition. They are terms corresponding to the third clause for *collection*, where either $S_1$ or $S_2$ could be empty collection. This can be easily trimmed by being type-assigned into **Utype** in the type/value assignment mapping $\mathcal{H}$. Also, the above definition automatically makes the empty collection, $<>$, as a polymorphic constant.

As the value constructors are also type constructors, it is easily to have SVP-types :

- **atom**.
- $tuple(T_1, \cdots, T_n)$, where each $T_i$ is a SVP type.
- $collection(T)$ if $T$ is a SVP type.

We note as a result of the principle of polymorphism where each type variable is universally quantified, homogeneity is automatically enforced as desired in the original definition.

Due to the fact that values of all those types are **SVP-values**, we add a universal type, namely **SVP-Utype**. Altogether, we have the Herbrand Type Universe as the set of SVP types together with the **SVP-Utype**. **SVP-Utype** is the least upper bound of the Herbrand Type Universe, and all non-**SVP-Utype** types are mutually disjoint in their denoted set of values. However, there is a slight pitfall due to the polymorphism of $<>$, since $\{<>\}$ is in the intersection of all types, namely, the greatest lower bound. As a common practice, we can address this by supposing that there are monomorphic incarnations of $<>$ in each type, and the polymorphic $<>$ is a shorthand without type subscript. Note this is necessary not only for theoretical simplicity, but also for securing the notion of homogeneity, since otherwise, $<>$ would be typed as **SVP-Utype** and $collection(T)$ would contain a non-universally quantified instance and thus no longer be polymorphic nor homogeneous. It would then contradict the original design. An alternative remedy is to restrict the empty collection, $<>$, from being a SVP value syntactically, while keeping it as a semantically polymorphic shorthand.

Another related observation is that type variables in the definition of SVP types can only range over SVP types, excluding **SVP-Utype**. This restriction is due to the introduction of **SVP-Utype** as a completion of SVP types. This has the effect on the unit collection $< v >$, since $v$ would be only typed as a non-**SVP-Utype**. Otherwise, a pitfall would follow regarding *collection*(**SVP-Utype**), in which homogeneity of elements no longer holds. Homogeneity becomes meaningless if every value is possible. Note this is the case for any untyped system where all values constitute a single type.

Meanwhile, we note that the set of all elements of non-**SVP-Utype** does not exhaust the Herbrand Value Universe. This means **SVP-Utype** is not redundant, and certain aspects of heterogeneity are encapsulated in **SVP-Utype**. To illustrate, notice we could have a set of SVP values, which is modeled as *collection(***SVP-Utype***)*, for instance, $< 1 > \diamond < 2 > \diamond < a > \diamond < b > \diamond < (a, 1) > \diamond << b > \diamond < (2, c) >>$ . These are intuitively not homogeneous collections, though useful in practice. While maybe of some generality, such collections suffer from badly behaved polymorphism due to use of **SVP-Utype** as a constituent type. This is of course a design choice.

#### 4.1.2 Types of SVP Transducers

With the above understanding in mind, and also with the definition of *SVP transducer* and the requirement of **compositionality**, we can proceed to the type specifications of functions involved in the general form of SVP transducer.

Since the function composition of SVP-transducers is beyond the original consideration of type structure, we have adopted conventions on type inference and specification, as in simple type theory.

- The type specification for transducers determines their argument and resultant types. N-ary functions are made positionwise type correspondent. This is implied by the requirement of compositionality.
- Each arm of a casewise definition receives the same type, and the condition receives type *Bool*. This is just a consistency condition for type specification.
- Since polymorphism is allowed, we extend simple type matching to type unification.
- Since there is no function abstraction, we can neglect rules for abstraction.

In section 3 of [SVP 92], SVP-transducers are characterized as the composition of one or more functions, each of which can be written in the following divide-and-conquer form :

$$\begin{array}{lllll} a. & f(S) & = & F(Q_0, \rho(S)) \\ b. & F(Q, <>) & = & id_\theta \\ c. & F(Q, <x>) & = & h(Q, x) \\ d. & F(Q, S_1 \diamond S_2) & = & F(Q, \rho(S_1)) \;\; \theta \;\; F(\delta(Q, S_1), \rho(S_2)) \end{array}$$

where $Q_0$ is an arbitrary fixed value, $\pi$ is either the identity mapping or an SVP-transducer, and $h, \theta$, and $\delta$ are arbitrary SVP mappings of two arguments.

Along with the composition requirement and input specification, we have the initial type assignment [2] :

$$\begin{array}{llll} S & : & & \rightarrow \; collection(T_0) \\ id_\theta & : & & \rightarrow \; S_0 \\ f & : & collection(T_1) & \rightarrow \; collection(S_1) \\ F & : & T_{Q0} \times collection(T_2) & \rightarrow \; collection(S_2) \\ \rho & : & collection(T_3) & \rightarrow \; collection(S_3) \\ h & : & T_{Q1} \times T_4 & \rightarrow \; collection(S_4) \\ \delta & : & T_{Q2} \times collection(T_5) & \rightarrow \; T_{Q3} \\ \theta & : & collection(T_6) \times collection(T_6) & \rightarrow \; collection(T_7) \end{array}$$

We come up with following constraints by type unification :

[2] The initial type specification of $f, F, h, \theta$ is due to the fact that all SVP-transducers are to be composable. This is a design principle, slightly different from the composion constraint within the equations. This design choice may be dropped in some SVP applications. But the solution would go through, except the more general type specification, $T$, instead of $collection(T)$, is enought for $f, F, h, \theta$. Together with the input being collections, both the domain and codomain receive the same type $collection(T)$. See also summary in the following subsection.

| | | | | |
|---|---|---|---|---|
| 1) | $collection(T_0)$ | $=$ | $collection(T_3)$ | *Input* |
| 2) | $S_0$ | $=$ | $collection(S_1)$ | $id_\theta, f, b.$ |
| 3) | $collection(S_1)$ | $=$ | $collection(S_2)$ | $f, F, a.$ |
| 4) | $collection(T_3)$ | $=$ | $collection(S_1)$ | $\rho, F, a.$ |
| 5) | $collection(T_1)$ | $=$ | $collection(T_3)$ | $f, \rho, a.$ |
| 6) | $collection(S_3)$ | $=$ | $collection(T_2)$ | $\rho, F, a.$ |
| 7) | $T_4$ | $=$ | $T_2$ | $h, F, collection, c.$ |
| 8) | $S_2$ | $=$ | $S_4$ | $F, h, c.$ |
| 9) | $T_6$ | $=$ | $S_2$ | $F, \theta, d.$ |
| 10) | $T_7$ | $=$ | $S_2$ | $\theta, F, d.$ |
| 11) | $T_5$ | $=$ | $T_2$ | $\delta, F, d.$ |
| 12) | $T_2$ | $=$ | $S_3$ | $\rho, F, d.$ |
| 13) | $T_{Q0}$ | $=$ | $T_{Q1}$ | *c.Identity* |
| 14) | $T_{Q0}$ | $=$ | $T_{Q2}$ | *d.Identity* |
| 15) | $T_{Q3}$ | $=$ | $T_{Q0}$ | *d.Output* |

We have a consistent solution, namely, all *collection* type variables are identified with the same type $T_0$, except $S_0$ with $collection(T_0)$ and all $T_{Qi}$'s are all identified too. In fact, this is the most general type, or principal type in some literature, compatible with the specification. Namely, other consistent type assignment defined over the Herbrand Type Universe, satisfying the specification, would be an instance of, or less general (or less polymorphic) than, $T_0$ .

As a result, we have not only validated that the proposed polymorphic scheme for *SVP transducers* is indeed well-defined, but also that it allows maximal polymorphism as long as it is homogeneous. As a matter of fact, since the type for $Q$ ( parameter or state variable) of the scheme is never constrainted except for type identification. This means an implication that, if we allow the type variables $T_{Qi}$ of $Q$ to range over higher-order transducers or mappings, there is still a consistent solution, as far as type consistency is concerned. A way to incorporate this observation is to extend the Herbrand Type Universe, argument positionally or by cartesian product with other higher-order type structures, say second order lambda calculus. This is a direction worth further inquiry.

### 4.1.3 Summary

We can summarize some of the results of applying type analysis to the SVP model so far, and suggest possible future modifications.

- We have at least validated the scheme of *SVP transducer* and its generality, based on the requirement of polymorphism and homogeneity together with principle of function composition from simple type theory applied on a generic Herbrand Type Universe.
- As a result, we have to restrict the type of $\theta$ to be

  $$collection(T) \times collection(T) \longrightarrow collection(T).$$

  Namely, $\theta$ has to be applied to and produce collections of homogeneous SVP values. This is in fact consistent with intuition about SVP transducers. As the solution indicates, this is all due to the design principle of composibility of SVP-transducers, rather than type constraints.
- On the other hand, we have to either revise some of the rows of instances of $\theta$ as listed in the article [SVP 92]. This is no problem at all even as a result of design choice, as we can just add a layer of *collection* construction and use *unitcollectionvalue()* to retrieve it. For instances, $min, max, sum$ as $unitcollectionvalue \circ min$, $unitcollectionvalue \circ max$ and $unitcollectionvalue \circ sum$ respectively.

- The type specification of $h$ also requires an output type of *collection*$(T)$. This is certainly proper, and cause no problem to adapt, similar to the case with $\theta$.
- With these observation, along with the fully unconstrainted type specification of $Q$, the extension of function space in the type structure has its proper demand. Yet, it could be an independent dimension as far as the type specification of SVP-transducers is concerned.

## 4.2 Another Application

Following the same method, we can derive the type assignment for :

$$\begin{array}{lll} append([], L, L) & :- & true. \\ append([A|As], Bs, [A|AsBs]) & :- & append(As, Bs, AsBs). \end{array}$$

with corresponding type signature :

$$\begin{array}{lll} append(X1, Y1, Z1) & :- & true. \\ append(X \times X2, Y2, X \times Z2) & :- & append(X2, Y2, Z2). \end{array}$$

by solving the following type equations :

$$\begin{array}{lll} []_{type} & = & X1 \\ X1 & = & X \times X2 \\ X2 & = & X1 \\ Y1 & = & Y2 \\ Y1 & = & Z1 \\ Z1 & = & X \times Z2 \\ Z2 & = & Z1 \end{array}$$

We have the solution :

$$\begin{array}{lll} X1 & = & \times^{*}(X) \\ X2 & = & \times^{*}(X) \\ Y1 & = & Y2 \\ & = & Z1 \\ & = & Z2 \\ & = & \times^{\infty}(X) \cup \mathbf{Utype} \\ & = & \mathbf{Utype} \end{array}$$

This matches our intuition, and justifies the type assignment of **Utype** to uninstantiated logical variables and typed use of cons(). Here from the solution, the second argument of *append*/3 could be an arbitrary value, including an infinite list, while the first argument has to be a finite list.

# 5 Conclusion and Further Research

Inspired by applications in stream processing, and lazy computation and infinite objects commonly seen in Logic Programming, we started out with points of dissatisfaction with conventional type schemes. Taking the position of **Types for The Prolog Computational Model**, aiming toward a type account of the dynamic essence of lazy computation and infinite objects by uninstantiated logical variables, we classified the Herbrand Value Universe into subsets, and designated them as denotations for types. We then formalized the notion, and went further to introduce notions of polymorphism and homogeneity into the apparently simple and monomorphic, yet complete join semilattice, the Herbrand Type Universe. Aspects of heterogeneity were

also discussed, with the coarsest heterogeneity being captured by the universal type **Utype**. This universal type not only greatly enriches the type structure, but also provide insights for its power and applications.

We then applied the framework to an analysis of the SVP model, a typical stream processing model with parallelism. This example illustrates notions and terms of our type framework. The framework also helped us to resolve initial questions on some Logic programs.

Looking back upon what the issue was and how we accounted for it, the problem could be phrased as finding a suitable type discipline for a nominal Herbrand Universe of constructors and functors. As nominal as it is, this universe itself is a class of values, where every term combination is a value. While it is useful for first order application, many troubles arise when we want to classify this universe into well behaved subclasses, namely to impose a type structure over it. The only inherent structure is how a term is constructed symbolically. Other than that, there is no domain-specific relation between terms. Of course, those relations are formulated in a different syntactic category, namely, predicates. However, most of the time, we would like to work on some domain with meanings whose value terms do relate. Among others, type related properties are natural and prominent ones. Polymorphism and homogeneity are the two additional principles preventing us from overrestriction, while admitting useful and general applications. This article serves as a try, and so far, it is successful as far as our application is concerned.

There are a couple of directions for further research :

First is to exploit the Herbrand Type Universe. Obviously the semi-lattice type structure is kind of simple, though, as it shows, it does allow study and applications of a broad spectrum, e.g. polymorphism, homogeneity, function composition, unification-based type inference, etc. As remarked, there is no type construction for function abstraction, which is a marked disadvantage with respect to common type systems, such as various second order $\lambda$-calculi. Also, our type system in use excludes almost any possibility of heterogeneity. In addition, type variables are only universally quantified and placed in prenex form. The lack of existential quantification leads to the difficulties for abstract data type and related notions as representation independence, and separation of specification and implementation. The followings are some ideas in extending our type scheme :

- We can permit a constructor to be both a value constructor and type constructor, as is the case in the SVP model. By doing this we gain the edge of isomorphism between type structure and value structure. However, blindly doing this may actually push us back into an untyped system, since everything would be of a single type. We need to restrict the extension only for major value constructions, and keep functors as interpreted or defined functions over value universe, yet not reduced to predicators.
- There could actually be more types than those in the Herbrand Type Universe, such as the first order **Utype** or other higher order types or the type of all types and etc. We may just need some means to denote and harness them into the established segment. This generalization suggests making a distinction between deep and shallow (or global and local) polymorphism. Roughly speaking, some function may only depend on a few layers of type construction, not down to the very bottom atomic values, as long as the computation can proceed. This is often the situation for lazy evaluation paradigm, where only certain layers of type information are concerned rather than whole type information of each value. For example, some processing can be specified as of type $list(list(X)) \mapsto nat$, where $X$ is don't-care type variable independent of any type structure, signifying that only two layers of list structure is necessary sufficient for its correctness. We refer to the polymorphism in such cases as shallow polymorphism, while in conventional cases as deep polymorphism where either ground type or underlying type structure matters.
- In responding to the fact that all type expressions denotes disjoint SVP-value subsets, we can in fact introduce type constructors for disjoint union among these types, say, to allow degrees of heterogeneity. Proper injectors and categorical accounts may be used too.

Second, as a companion of richer type structure, we may like to consider the type inference rules. For now, the type inference is very simple, needing only unification. Richer structure certainly leads to more complicated type inference. Also, the **Types Is Sets** position leaves open other methods. It would be interesting to examine if, say, **A Type is A Formula**, could be combined with Logic Programming.

# References


[BD 83] Joham van Benthem and Kees Doets, *Higher Order Logic*, in D. Gabbay and F. Guenthner eds, *Handbook of Philosophical Logic*, Vol. 1, D. Reidel Publishing Company, 1983.

[Ch 40] Alonzo Church, *A Formulation of The Simple Theory of Types*, Journal of Symbolic Logic, 5, 1940, pp. 56-68.

[CW 85] L. Cardelli and P. Wegner, *On Understanding Types, Data Abstraction, and Polymorphism*, ACM Computing Surveys, 17(4), 1985, pp. 471-522.

[FLO 83] Steven Fortune, Daniel Leivant and Michael O'Donnell, *The Expressiveness of Simple and Second-Order Type Structures*, JACM, 30(1), Jan. 1983, pp. 151-185.

[GM 84] J.A. Goguen and J. Meseguer, *Equality, Types, Modules and (Why Not ?) Generics For Logic Programs*, J. of Logic Programming, Vol 2 (1984), pp.179-210.

[Ll 87] John Wylie Lloyd, *Foundations of Logic Programming*, Spring-Verlag, 1987.

[Ho 80] W. Howard, *The Formula-As-Types Notion of Construction*, in : *To H.B. Curry : Essays on Combinatory Logic, Lambda-Calculus and Formalism*, Academic Press, 1980, pp. 479-490.

[Ma 73] Per Martin-Löf, *An Intuitionistic Theory of Types : Predicative Part*, in : *Logic Colloquium, '73*, North-Holland, 1973, pp.73-118.

[Mi 84] P. Mishra, *Toward A Theory of Types in Prolog*, in : *International Symposium on Logic Programming*, IEEE 1984, pp. 289-298.

[Mi 90] John C. Mitchell, *Type Systems for Programming Language*, in *Handbook of Theoretical Computer Science*, by J. van Leeuwen, (eds.), Vol.B, 1990.

[SS 87] Leon Sterling and Ehud Shapiro, *The Ar of Prolog*, The MIT Press, 1987.

[SVP 92] D. Stott Parker,Eric Simon and Patrick Valduriez, *SVP - A Model Capturing Sets, Streams, and Parallelism*, UCLA, Computer Science Department Technical Report,CSD-920020, April 1992. To appear, *Intnl. Conf. on Very Large Data Bases*, Vancouver, Aug, 1992.

[TZ 92] J.V. Tucker and J.I. Zucker, *Deterministic and Nondeterministic Computation, and Horn Programs, On Abstract Data Types*, Journal Of Logic Programming, 13(1), May 1992, pp. 23-56.

[XW 88] J. Xu and D.S. Warren, *A Type Inference System for Prolog*, in : *Proceedings of the 5th International Conference and Symposium on Logic Programming*, Aug. 1988, pp. 604-619.

[YS 91] Eyal Yardeni and Ehud Shapiro, *A Type System for Logic Programs*, Journal Of Logic Programming, 10(2), Feb. 1992, pp. 125-154.